TESTE DE *SOFTWARE*

AULA 3

Prof.ª Maristela Weinfurter

CONVERSA INICIAL

Neste conteúdo, vamos abordar assuntos importantes e complementares, tais como testes unitários e uso da ferramenta JUnit, testes de componentes, TDD como *framework* de desenvolvimento de *software* orientado a testes, boas práticas em testes e testes colaborativos.

Todas essas são algumas práticas e ferramentas que em muito auxiliam nos processos de testes, avaliações se verificações, garantindo que nosso *software* esteja dentro de padrões e processos mais maduros.

Há uma quantidade grande de *frameworks*, técnicas e ferramentas para testes de *software*. Essa área da engenharia de *software* é complexa e por vezes difícil de ser implementada, uma vez que as escolhas são difíceis em decorrência da variedade de opções.

As ideias-chave do teste de *software* são aplicáveis independentemente do software envolvido e de qualquer metodologia de desenvolvimento específica (clássico, ágil, ou qualquer outro tipo de modelo).

Testes de componentes são nosso primeiro desafio e estes levam em consideração a observação de toda a lógica do domínio bem como das regras de negócios. Os critérios de erros e outros aspectos também são elementos importantes quando pensamentos em componentes.

Na sequência, vamos explorar o mundo TDD, que preza pela construção de casos de testes que vão orientar todo nosso desenvolvimento. Na sequência, numa aproximação real entre o TDD e o XP, temos o conceito de testes unitários, seguido pela amostra de uma ferramenta muito utilizada pelos desenvolvedores Java.

Finalizamos com boas práticas de testes, assim como temos boas práticas em desenvolvimento de *software* e finalizamos com testes colaborativos. Aliás, o grande segredo do sucesso de quaisquer projetos de desenvolvimento ou testes de *software* é justamente a harmonia na comunicação e na colaboração entre os vários times envolvidos no processo de testes.

TEMA 1 – TESTES DE COMPONENTE

Antes de entrarmos no assunto testes de componentes propriamente, é importante relembrarmos sobre os testes de serviços que tratam de toda a lógica

do domínio e das regras de negócios, dos critérios de erros, dos mecanismos de repetição, do armazenamento de dados, entre outros.

Os testes de serviço às vezes residem em uma base de código separada, mas para obter *feedback* rápido, é melhor mantê-los como parte dos próprios componentes de serviço. Eles são um pouco mais complexos de criar e manter do que os testes de unidade, pois envolvem uma configuração real de dados de teste no banco de dados. Normalmente, os testadores da equipe possuem esses testes. Eles são executados mais rapidamente do que os testes de ponta a ponta orientados pela interface do usuário e um pouco mais lentos do que os três testes de micronível anteriores (testes de unidade, integração e contrato). Ferramentas como *REST Assured*, *Karate* e *Postman* podem ser usadas para automatizar testes de API (Mohran, 2022).

Qualquer entidade que esteja bem encapsulada e possa ser reutilizada ou substituída independentemente, como um serviço, é chamada de *componente*. Quando ouvimos o termo *testes de componentes*, podemos pensar em testes de serviço como um exemplo.

Na maioria dos aplicativos da *web* de médio e grande porte, existem alguns pontos de integração com componentes internos ou externos, como serviços, interface do usuário, bancos de dados, *caches*, sistemas de arquivos e assim por diante, que podem ser distribuídos pelos limites da rede e da infraestrutura. Para testar se todos esses pontos de integração funcionam conforme o esperado, precisamos escrever testes de integração executados nos sistemas de integração reais. O foco de tais testes deve ser verificar os fluxos de integração positivos e negativos, não a funcionalidade detalhada de ponta a ponta. Como resultado, eles devem ser idealmente tão pequenos quanto os testes de unidade.

No exemplo do aplicativo de comércio eletrônico, o serviço de pedido se integra a componentes internos, como a IU de comércio eletrônico, o banco de dados e outros serviços para troca de informações. Ele também se integra ao serviço PIM do fornecedor externo e aos sistemas *downstream*. Precisamos escrever testes de integração para cada serviço para verificar se ele pode se comunicar adequadamente com outros serviços dependentes e com o banco de dados e ver se os testes de integração de serviço de ordem devem ser adicionados para verificar a integração com esses sistemas e serviços externos em particular.

Qualquer entidade que esteja encapsulada e possa ser reutilizada ou substituída independentemente, como um serviço, é chamada de *componente*.

O teste de componentes envolve a verificação sistemática dos componentes de nível mais baixo na arquitetura de um sistema. Dependendo da linguagem de programação usada para criá-los, esses componentes têm vários nomes, como *unidades*, *módulos* ou (no caso de programação orientada a objetos) *classes*. Os testes correspondentes são, portanto, chamados de *testes de módulo*, *testes de unidade* ou *testes de classe*.

Independentemente de qual linguagem de programação é usada, os blocos de construção de *software* resultantes são os *componentes*, e os testes correspondentes são chamados de *testes de componentes*. Os requisitos específicos do componente e o projeto do componente (ou seja, suas especificações) devem ser consultados para formar a base de teste. Para projetar testes de caixa branca ou avaliar a cobertura do código, devemos analisar o código-fonte do componente e usá-lo como base de teste adicional. No entanto, para julgar se um componente reage corretamente a um caso de teste, consultamos a documentação do projeto ou dos requisitos. Os módulos, unidades ou classes são objetos de teste típicos. No entanto, coisas como *scripts* de *shell*, *scripts* de banco de dados, procedimentos de migração e conversão de dados, conteúdo de banco de dados e arquivos de configuração também podem ser objetos de teste. Um teste de componente verifica a funcionalidade interna de um componente

Um teste de componente normalmente testa apenas um único componente isoladamente do resto do sistema. Esse isolamento serve para excluir influências externas durante o teste: se um teste revelar uma falha, é obviamente atribuível ao componente que estamos testando. Também simplifica o *design* e a automação dos casos de teste, devido ao seu escopo restrito.

Em tempo de execução, um componente de *software* precisa interagir e trocar dados com vários componentes vizinhos, e não pode ser garantido que o componente não será acessado e usado de forma incorreta. Nesses casos, o componente endereçado incorretamente não deve simplesmente parar de funcionar e travar o sistema, mas deve reagir de forma razoável e robusta. O teste de robustez é, portanto, outro aspecto importante do teste de componentes. O processo é muito semelhante ao de um teste funcional comum, mas atende ao componente em teste com dados de entrada inválidos em vez de dados

válidos. Esses casos de teste também são chamados de *testes negativos* e assumem que o componente produzirá tratamento de exceção adequado como saída. Se o tratamento de exceção adequado não estiver integrado, o componente poderá produzir erros de tempo de execução, como divisão por zero ou acesso de ponteiro nulo, que causam o travamento do sistema.

O teste de componentes é altamente orientado ao desenvolvimento. O testador geralmente tem acesso ao código-fonte, suportando uma técnica de teste orientada a caixa branca em testes de componentes. Podemos projetar casos de teste usando o conhecimento existente da estrutura interna, métodos e variáveis de um componente. A disponibilidade do código-fonte também é uma vantagem durante a execução do teste, pois conseguimos usar ferramentas de depuração apropriadas para observar o comportamento das variáveis durante o teste e ver se o componente funciona corretamente ou não. Um depurador também permite manipular o estado interno de um componente, para que possamos iniciar exceções deliberadamente quando estiver testando a robustez.

Durante a integração, os componentes individuais são cada vez mais combinados em unidades maiores. Essas unidades integradas podem já ser muito grandes para inspecionar o código completamente. Determinar se o teste de componente é feito em componentes individuais ou em unidades maiores é uma decisão importante que deve ser tomada como parte do processo de integração e planejamento de teste.

Test-first é a abordagem de última geração para teste de componentes (e, cada vez mais, em níveis de teste mais altos também). A ideia é primeiro projetar e automatizar seus casos de teste e programar o código que implementa o componente como uma segunda etapa. Essa abordagem é fortemente iterativa: testamos o código com os casos de teste que já projetamos e, em seguida, estendemos e melhoramos o código do produto em pequenas etapas, repetindo até que o código cumpra seus testes. Esse processo é chamado de *programação test-first* ou *desenvolvimento orientado a testes* (geralmente abreviado para TDD). Se derivamos os casos de teste sistematicamente usando técnicas de *design* de teste bem fundamentadas, essa abordagem produzirá ainda mais benefícios – por exemplo, testes negativos também serão elaborados antes começarmos a programar – e a equipe é forçada a esclarecer o objetivo pretendido. comportamento do produto para esses casos.

É possível deixar completamente de fora o teste de componentes e iniciar o processo de teste diretamente com o teste de integração, inclusive sendo uma prática comum. No entanto, esta abordagem tem desvantagens potencialmente graves:

- A maioria das falhas reveladas por esse tipo de teste será causada por falhas funcionais dentro de componentes individuais. Em outras palavras, o que é realmente um teste de componente é executado em um ambiente inadequado que complica o acesso a componentes individuais;
- Como não há acesso fácil a cada componente individual, algumas falhas não serão provocadas e, portanto, impossíveis de encontrar;
- Se ocorrer uma falha ou travamento, é difícil ou impossível localizar o componente que causou a falha;
- Ficar sem o teste de componentes economiza esforço apenas ao preço de baixas taxas de descoberta de falhas e maiores esforços de diagnóstico.

Combinar testes de componentes com testes de integração é muito mais eficiente.

TEMA 2 – TDD – *TEST DRIVE DEVELOPMENT*

O TDD (*Test Driven Development*) é uma concepção ágil para o desenvolvimento de *software* orientado a testes. Neste, os desenvolvedores escrevem seus casos de testes e depois programam as funcionalidades. O TDD encoraja que o projeto do código deva ser simples e inspire confiança. O livro mais completo e original que aborda o TDD foi escrito por Kent Beck. O TDD é um conceito atrelado à metodologia ágil XP (*Extreme Programming*). O TDD baseia-se pequenos ciclos repetitivos, por meio da criação da funcionalidade do *software* a partir de um teste unitário. Todo processo inicial falha, e a cada nova linha de código implementada, rodamos novamente o teste, até que ele fique totalmente sem erros. TDD caminha lado a lado com as boas práticas de desenvolvimento de *software*, para garantir código limpo, menos acoplado e mais coeso.

Com TDD conseguimos:

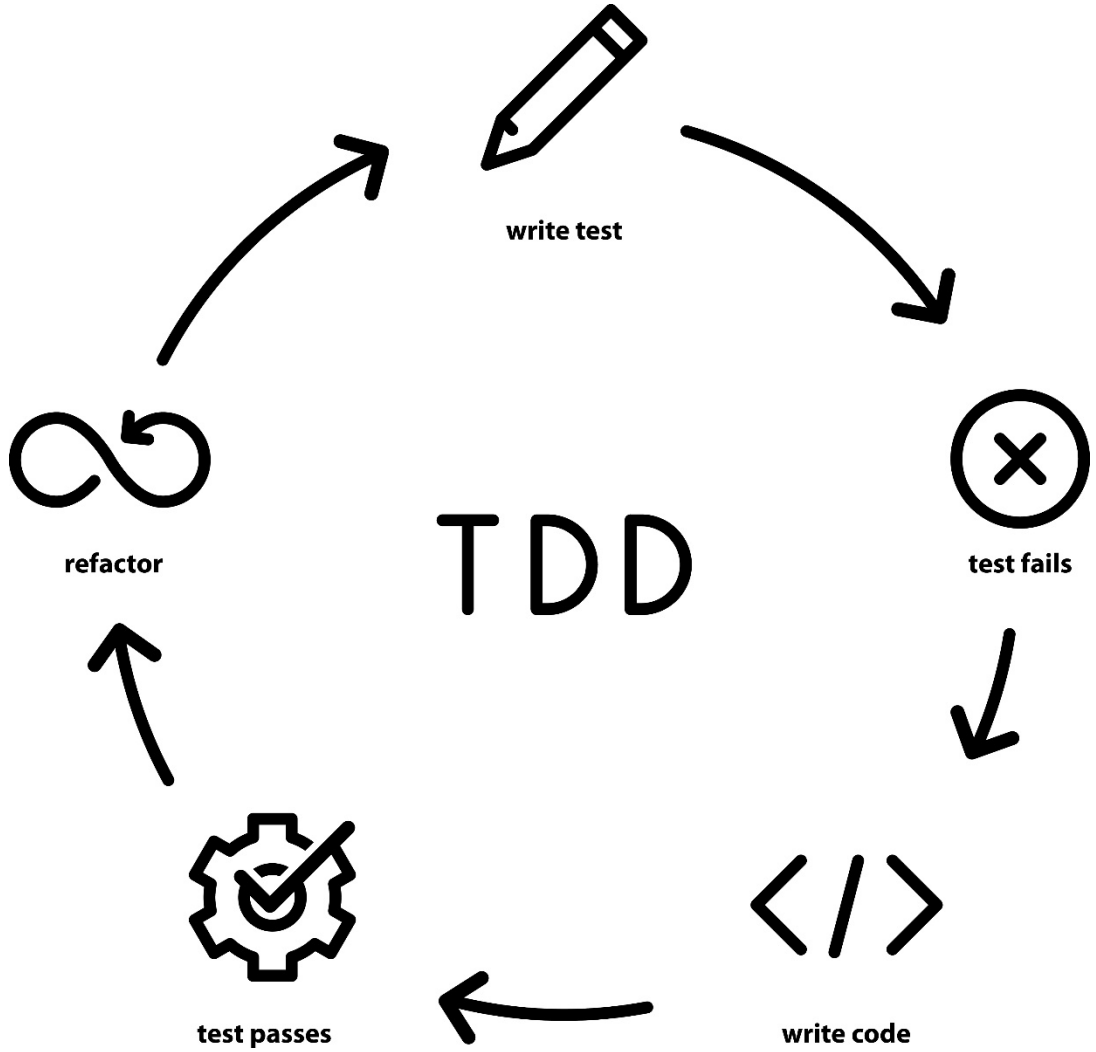
- código limpo (sem código desnecessário e/ou duplicado);
- código fonte dos testes como documentação dos casos de testes;

- código confiável, logo com mais qualidade;
- suporte para teste de regressão;
- ganho de tempo na depuração e correção de erros;
- desenvolvimento refatorado constantemente;
- e baixo acoplamento do código.

O Ciclo de Desenvolvimento (figura 1) é composto por sinalizadores *red*, *green* e *refactor*.

1. Escrita do teste inicial. *Flag red*;
2. Adição de nova funcionalidade;
3. Execução do teste passar. *Flag green*;
4. Refatoração do código;
5. Escrita do próximo teste.

Figura 1 – Ciclo de desenvolvimento do TDD



Crédito: Vector Street/Shutterstock.

Com a implantação do TDD, o *feedback* ágil e a entrega de novas funcionalidades e sem quebras são garantidos. Para o TDD, as quebras são os problemas que ocorrem quando um teste não passa. Os testes unitários orientam o desenvolvimento simples e refatorado de forma atômica e iterativa. Com isso, a produtividade do desenvolvedor, inicialmente parece burocrática, pois exige que este mude sua forma de pensamento e crie primeiramente os testes. Conforme o desenvolvedor adquire experiência, o processo de desenvolvimento também é mais rápido e produtivo. Esses testes nos ajudam a não tomarmos caminhos errados.

Em primeiro momento, o desenvolvedor tem a sensação de que algo está sem lógica, pois criar o código de testes sem ao menos termos uma linha de código implementada parece algo totalmente estranho. Mas é justamente o que Beck (2010) propõe: que o primeiro teste sempre irá falhar, porque o teste não encontrará código algum. Logo depois da primeira escrita de um pequeno teste unitário, e que pode não estar totalmente completo também, a escrita do código da funcionalidade é criada com mais clareza e segurança pelo desenvolvedor. A isto chamamos de *KISS* (*Keep it Simple, Stupid*), que conduz à escrita de um código limpo, simples e funcional. Inevitavelmente, ao utilizarmos boas práticas e padrões de programação, garantimos que nossa refatoração caminhe da forma mais correta possível.

Uma vez superada a técnica do TDD, passamos agora de forma mais confortável para refatoração. Depois que fizemos o teste passar, é o momento de refatorar, retirando duplicidades, melhorando as nomenclaturas, criando métodos e classes e utilizando os padrões necessários. Junto à refatoração, é hora também de observarmos que cada classe deve ter uma única responsabilidade, a isto chamamos de SRP (*Single Responsibility Principle*).

2.1 TDD e *Extreme Programming*

Quando falamos em *Extreme Programming* (XP), estamos dizendo que precisamos pensar “fora da caixa” ao falarmos sobre hábitos e padrões clássicos de desenvolvimento de *software*. Precisamos focar na produtividade saudável. Para compreender e usar o XP, é necessário o uso de técnicas e bons relacionamentos com nossos *coworkings*.

XP é um estilo de programação que foca na excelência do *software* por meio de técnicas de programação aliado a uma comunicação clara e sem ruídos entre os desenvolvedores.

Com a aplicação do XP em nosso dia a dia, podemos observar que:

- Conquistamos uma filosofia de desenvolvimento de software baseada nos valores de comunicação, *feedback*, simplicidade, coragem e respeito;
- Utilizamos um conjunto de práticas comprovadamente úteis para melhorar o desenvolvimento de software. As práticas se complementam, amplificando seus efeitos. Eles são escolhidos como expressões dos valores;
- Criamos um conjunto de princípios complementares, técnicas intelectuais para traduzir os valores em prática, úteis quando não há uma prática útil para o seu problema específico;
- E finalmente, desenvolvemos uma comunidade que compartilha esses valores e muitas das mesmas práticas.

XP é um caminho de melhoria para a excelência da sinergia entre pessoas que se unem para desenvolver *software*. São detalhes importantes que a distinguem de outras metodologias:

- Seus ciclos de desenvolvimento curtos, resultando em *feedback* precoce, concreto e contínuo;
- Sua abordagem de planejamento incremental, que rapidamente apresenta um plano geral que deve evoluir ao longo da vida do projeto;
- Sua capacidade de agendar com flexibilidade a implementação da funcionalidade, respondendo às necessidades de negócios em constante mudança;
- Sua dependência de testes automatizados escritos por programadores, clientes e testadores para monitorar o progresso do desenvolvimento, permitir que o sistema evolua e detectar defeitos antecipadamente;
- Sua dependência de comunicação oral, testes e código-fonte para comunicar a estrutura e a intenção do sistema;
- Sua dependência de um processo de *design* evolutivo que dura tanto quanto o sistema dura;
- Sua confiança na estreita colaboração de indivíduos ativamente engajados com talento comum;

- Sua confiança em práticas que trabalham tanto com os instintos de curto prazo dos membros da equipe quanto com os interesses de longo prazo do projeto.

De acordo com Kent (2004), “O XP é uma metodologia leve para equipes pequenas e médias que desenvolvem *software* diante de requisitos vagos ou que mudam rapidamente”. No XP, construímos *software* que faz o necessário para criar valor para o cliente. O corpo de conhecimento técnico necessário para ser uma equipe de destaque é grande e crescente. Ela trata de restrições no desenvolvimento de *softwares*. Não aborda gerenciamento de portfólio de projetos, justificativa financeira de projetos, operações, *marketing* ou vendas. O XP tem implicações em todas essas áreas, mas não aborda essas práticas diretamente. *Metodologia* é frequentemente interpretada como um conjunto de procedimentos e regras a serem seguidos e teremos sucesso no final, mas infelizmente, metodologias não funcionam como programas. As pessoas não são computadores. Cada equipe faz o XP de maneira diferente com graus variados de sucesso.

Em XP, podemos trabalhar com times de quaisquer tamanhos e de ampla variedade de projetos. Os valores e princípios por trás do XP são aplicáveis em qualquer escala. As práticas precisam ser aumentadas e alteradas quando muitas pessoas estão envolvidas. Ele ainda, se adapta a requisitos vagos ou que mudam rapidamente, o que é extremamente necessário num cenário de empresas que mudam rapidamente para se adaptarem às novas realidades do mercado. Porém, isso não o restringe a esse cenário.

Com o XP reconciliamos a humanidade dos desenvolvedores com sua produtividade como a boa prática de desenvolvimento de *software*. A chave para o sucesso encontra-se na aceitação de que somos pessoas em um negócio de pessoa para pessoa e que a técnica também importa, pois somos pessoas técnicas em um campo técnico. Olhando para os processos, existem maneiras melhores e piores de trabalhar, e a busca da excelência na técnica é fundamental. A técnica apoia as relações de confiança, logo, se pudermos estimar com precisão nosso trabalho, e entregar qualidade na primeira vez, criaremos ciclos de *feedback* rápidos, tornando-nos assim um desenvolvedor XP confiável. No XP, todos os participantes aprendem um alto nível de técnica a serviço dos objetivos do time como um todo.

Os times XP jogam para vencer e aceitam a responsabilidade pelas consequências. Quando a autoestima não está vinculada ao projeto, somos livres para fazer o nosso melhor trabalho em qualquer circunstância. Manter um pouco de distância nos relacionamentos, reter o esforço por falta ou excesso de trabalho, adiar o *feedback* para outra rodada de difusão de responsabilidade: nenhum desses comportamentos tem lugar em um time XP.

Quando pensamos em qualidade de *software*, o XP é uma disciplina de desenvolvimento de software que aborda o risco em todos os níveis do processo de desenvolvimento. Além de ser produtivo, produz *software* de alta qualidade e é muito divertido de executar.

Abordagem dos riscos no processo de desenvolvimento com XP:

- **Agendamentos:** O XP exige ciclos de lançamento curtos, alguns meses no máximo, portanto o escopo de qualquer atraso é limitado. Dentro de uma versão, usa iterações de uma semana de recursos solicitados pelo cliente para criar um *feedback* detalhado sobre o progresso. Dentro de uma iteração, planeja com tarefas curtas, para que a equipe possa resolver problemas durante o ciclo. Por fim, o XP exige a implementação dos recursos de prioridade mais alta primeiro, portanto, quaisquer recursos que passarem do lançamento terão um valor menor;
- **Projeto cancelado:** o XP pede que a parte do time voltada para os negócios escolha a menor versão que faça mais sentido para os negócios, para que haja menos erros antes da implantação e o valor do *software* seja maior;
- **O sistema literalmente dá problema**: o XP cria e mantém um conjunto abrangente de testes automatizados, que são executados e executados novamente após cada alteração (muitas vezes ao dia) para garantir uma linha de base de qualidade. Além de sempre manter o sistema em condições de implantação, para o qual os problemas não podem se acumular;
- **Taxa de defeitos-testes:** o XP tem a perspectiva de programadores escrevendo testes (função por função) e de clientes escrevendo testes (programa-recurso-por-programa-recurso);
- **Negócios incompreendidos:** o XP exige que pessoas orientadas para negócios sejam membros de primeira classe do time. A especificação do

projeto é continuamente refinada durante o desenvolvimento, para que o aprendizado do cliente e da equipe possam ser refletidos no *software*;

- **Mudanças nos negócios:** o XP reduz o ciclo de lançamento, portanto, há menos mudanças durante o desenvolvimento de um único lançamento. Durante uma versão, o cliente pode substituir uma nova funcionalidade por uma funcionalidade ainda não concluída. A equipe nem percebe se está trabalhando em funcionalidades recém-descobertas ou recursos definidos anos atrás;
- **Falso rico em recursos:** o XP insiste que apenas as tarefas de prioridade mais alta sejam abordadas;
- **Rotatividade de pessoal**: o XP pede aos programadores que aceitem a responsabilidade de estimar e completar seu próprio trabalho, dá-lhes *feedback* sobre o tempo real gasto para que suas estimativas possam melhorar e respeita essas estimativas. As regras para quem pode fazer e alterar estimativas são claras. Assim, há menos chance de um programador ficar frustrado ao ser solicitado de fazer algo impossível. XP também incentiva o contato humano entre a equipe, reduzindo a solidão que muitas vezes está no centro da insatisfação no trabalho. Finalmente, o XP incorpora um modelo explícito de rotatividade de pessoal. Os novos membros da equipe são incentivados a aceitar gradualmente mais e mais responsabilidades e são auxiliados ao longo do caminho uns pelos outros e pelos programadores existentes.

A metodologia XP pressupõe que sejamos parte de uma equipe, idealmente com metas claras e um plano de execução. Também define que os times trabalhem em pares. Dentro da metodologia, a mudança deve ser barata e presume que temos a aspiração de crescimento e melhoria de nossas habilidades e relacionamentos, bem como estarmos dispostos a atingirmos esses objetivos.

Uma coisa bem interessante no TDD é a integração contínua. Com os testes sempre funcionando, isso garante que o ciclo de programação se torne rápido e eficiente. Quanto aos projetos, quanto mais simples, mais fáceis de continuidade e de testes, podendo ser automatizado. Já a refatoração é uma forma de simplificar quaisquer comportamentos do código. Gera maior confiança ao time de desenvolvimento na escrita da próxima rodada de testes e de código.

Finalmente, a entrega contínua é garantida de forma mais eficiente com o uso do TDD. Obviamente, como qualquer técnica, possui suas limitações. Não há como testar UIs automaticamente com TDD, nem de código de terceiros e há ainda uma limitação quanto aos testes relacionados a bancos de dados.

Aprendemos que TDD é uma técnica muito interessante e pode auxiliar em muito na construção de código limpo, organizado, com padrões e testado. Mas como toda e qualquer técnica, sempre há a curva de aprendizado e adaptação pelo time de trabalho. É importante que o time esteja maduro em relação à compreensão do motivo de se utilizar técnicas, que no início parecem mais atrapalhar do que ajudar. Mas é justamente a curva de aprendizado e amadurecimento que falamos. Para ganharmos produtividade e melhorias em qualidade, vamos ter sempre um primeiro passo que nos parece não sair do lugar. Mas os resultados a médio e longo prazo sempre são bons. Tudo é uma questão de unanimidade e persistência do time.

TEMA 3 – JUnit E DEMAIS FERRAMENTAS

Teste unitário é a fase de teste de cada unidade do *software*. O objetivo neste momento é o isolamento de cada parte do *software* com a ideia de garantir que cada pequena parte esteja funcionando conforme o especificado.

Esse tipo de teste, assim como todos os demais, carece de um bom planejamento. Logo, o desenvolvedor deve fazer a avaliação sempre que pensar nos requisitos para cada funcionalidade a ser testada e em quais entradas e saídas queremos diante do processamento do fluxo dos dados. Normalmente conhecido como *Unit*, possui uma estrutura de testes automáticos unitários e consiste na verificação da menor unidade do projeto de *software*.

O *Unit Test* é de responsabilidade dos desenvolvedores durante o processo de implementação do código. Ou seja, após a programação de uma classe, deve-se executar um teste unitário. No entanto, mesmo sendo responsabilidade de um *dev*, um QA deve estar comprometido na criação em conjuntos de testes unitários para contribuição do melhor desempenho do *software*. Sem robustez nos testes, a técnica falhará.

A redução da quantidade de *bugs* é considerável ao longo da implementação do *software*. Estes funcionam através da comparação de resultados esperados das funcionalidades com o código escrito.

Os principais objetivos do teste de unidade são verificar se um aplicativo funciona conforme o esperado e detectar bugs antecipadamente. Embora os testes funcionais nos ajudem a atingir os mesmos objetivos, os testes de unidade são extremamente poderosos e versáteis e oferecem muito mais.

Objetivos dos testes de unidade:

- Permitir maior cobertura de teste do que testes funcionais;
- Aumente a produtividade da equipe;
- Detecte regressões e limite a necessidade de depuração;
- Dê-nos a confiança para refatorar e, em geral, fazer alterações;
- Melhorar a implementação;
- Documentar o comportamento esperado;
- Ativar cobertura de código e outras métricas.

O primeiro tipo de teste que qualquer aplicativo precisa ter é o teste unitário. Se tivermos que escolher entre escrever testes unitários e testes funcionais, devemos focar na escrita do segundo tipo. Em nossa experiência, os testes funcionais cobrem cerca de 70% do código do aplicativo. Para irmos além e fornecermos mais cobertura de teste, devemos escrever testes de unidade.

Os testes de unidade podem simular facilmente condições de erro, o que é extremamente difícil para os testes funcionais (e impossível em alguns casos). Os testes de unidade também fornecem muito mais do que apenas testes, conforme explicado nas seções a seguir. Esses testes auxiliam no aumento da produtividade da equipe, na detecção e regressões e limitação da depuração e no refatoramento de código com mais confiabilidade.

Os principais objetivos do teste de unidade são o de verificação se o aplicativo funciona conforme o esperado e na detecção de *bugs* antecipadamente. Embora os testes funcionais nos ajudem a atingir os mesmos objetivos, os testes de unidade são extremamente poderosos e versáteis e oferecem muito mais.

Lembrando que os testes de unidade:

- Permitem maior cobertura de teste do que testes funcionais
- Aumentam a produtividade da equipe
- Detectam regressões e limite a necessidade de depuração
- Atribuem confiança para refatoração e alterações
- Melhoria na implementação

- Documentação do comportamento esperado
- Ativação da cobertura de código e outras métricas.

É algo que vem se tornando cada dia mais elementar dentro da programação e com isto, várias linguagens já possuem suas ferramentas de automatização de testes unitários, tais como:

1. Unit Testing Framework, Pytest e Locust para linguagem Python.
2. XCTest para Swift.
3. Test::Unit, RSpec e Minitest para Ruby.
4. Mocha, Jasmine, Jest, Protractor e Qunit para JavaScript.
5. PHPUnit para PHP.
6. NUnit para C#.
7. JUnit para Java.

Geralmente os testes de *software* são pensados antes da implementação do código, pois eles acabam nos orientando no desenvolvimento de cada classe do *software*.

3.1 JUnit

JUnit é considerado uma API (código aberto) por alguns e um *framework* por outros, o qual faz a criação de testes unitários em linguagem de programação JAVA. O autor do livro *Design patterns*, Erich Gamma, foi um dos idealizadores da JUnit, a qual deve permitir a criação de testes de fácil escrita e execução. Erich Gama, juntamente com Kent Beck (2003), prepararam a ideia do teste unitário com a intenção de automatizá-lo. O JUnit possibilita o teste de um componente de forma isolada, definindo chamadas de métodos que, por meio de uma simulação de entrada geram a validação das saídas esperadas. Nesse caso, as entradas são parâmetros dos métodos e as saídas são o retorno, que podem ser o estado do objeto ou exceções.

Assert.assertEquals(201, meuMetodo(105));

A expressão anterior faz a validação do método meuMetodo() para o resultado 201. O parâmetro do método é 105. Podemos afirmar que JUnit é um teste do tipo caixa branca, pois facilita a correção de métodos e objetos.

Dentre as vantagens do JUnit, encontram-se:

- Criação rápida de testes com aumento de qualidade do código em desenvolvimento;
- Criação de uma hierarquia dos testes;
- Software livre e código aberto;
- Ganho de tempo no desenvolvimento ao diminuir o tempo de depuração
- Conhecida por um grande número de desenvolvedores;
- Validação dos resultados dos testes com resposta imediata.

Segundo o tutorial do JUnit (S.d.), O JUnit é organizado da seguinte forma:

> 1. Existe uma classe *Test* que contém um método runTest responsável por fazer testes particulares.
> 2. Duas classes que herdam da classe Test. A classe TestCase, que testa os resultados de um método, e a classe TestSuite que faz um único teste em mais de um método registrando os resultados.
> 3. A classe TestCase possui também os métodos *setUp* **e** *TearDown*.

3.1.1 Método de comparação – Assertequals()

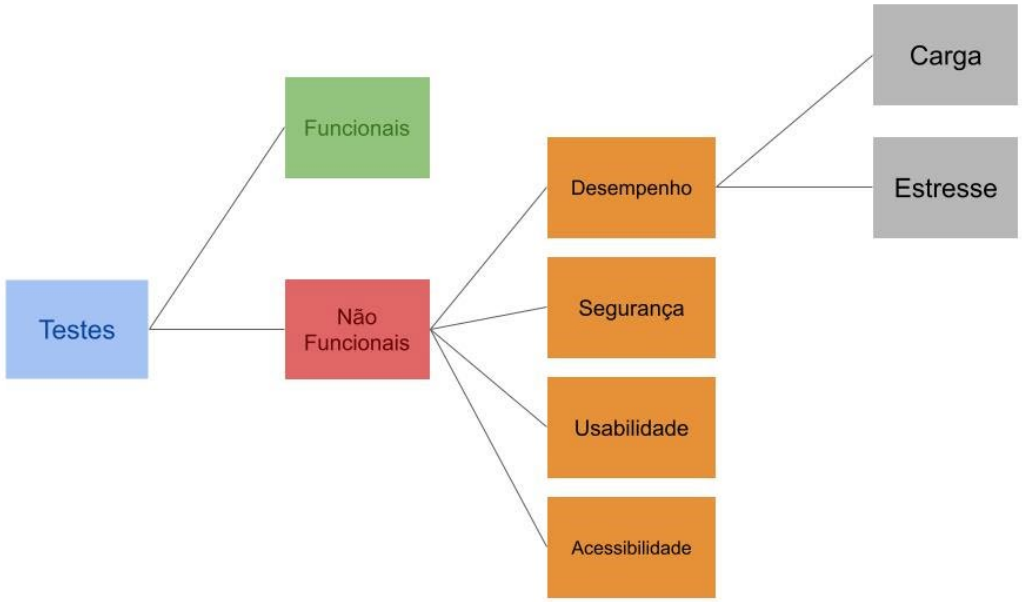
O método assertEquals é um método que pode ser implementado de várias formas diferentes. Ele recebe como parâmetro o resultado do método que está sendo testado e o resultado esperado pelo desenvolvedor caso o método testado esteja correto. O tipo desses valores passados como parâmetro pode ser vários (*int, double, String* etc…).

3.1.2 Métodos SetUp() e tearDown()

O método setUp() é utilizado para sinalizar o início do processo de teste. Vem antes do método de teste. O método tearDown() sinaliza o final do processo, desfazendo o que o setUp() fez. Vem depois do método de Teste.

Considerando uma classificação para casos de teste, classificamos todos os testes como *funcionais* e *não funcionais* e, assim, podemos pensar em alguns níveis de testes não funcionais, tais como:

Figura 2 – Taxonomia para testes funcionais e não funcionais



Fonte: Garcia, 2017.

Com base nessa proposta de esquema de taxonomia, vamos criar nossas meta-anotações personalizadas para folhas dessa estrutura de árvore: @Functional, @Security, @Usability, @Accessiblity, @Load e @Stress. Observe que em cada anotação estamos utilizando uma ou mais anotações @Tag, dependendo da estrutura previamente definida. Primeiro, podemos ver a declaração de @Functional:

```
package io.github.bonigarcia;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.junit.jupiter.api.Tag;

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("functional")
public @interface Functional {
}
```

Fonte: Garcia, 2017.

Em seguida, definimos a anotação @Security com *tags* não funcionais e de segurança:

```
package io.github.bonigarcia;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.junit.jupiter.api.Tag;

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("non-functional")
@Tag("security")
public @interface Security {
}
```

Fonte: Garcia, 2017.

Da mesma forma, definimos a anotação @Load, mas desta vez marcando com *non-functional*, *performance* e *load*:

```
package io.github.bonigarcia;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.junit.jupiter.api.Tag;

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("non-functional")
@Tag("performance")
@Tag("load")
public @interface Load {
}
```

Fonte: Garcia, 2017.

Por fim, criamos a anotação @Stress (com *tags* não funcionais, *performance* e *stress*):

```
package io.github.bonigarcia;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.junit.jupiter.api.Tag;

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("non-functional")
@Tag("performance")
@Tag("stress")
public @interface Stress {
}
```

Fonte: Garcia, 2017.

Agora, podemos usar nossas anotações para marcar (e depois filtrar) os testes. Por exemplo, no exemplo a seguir estamos usando a anotação @Functional no nível da classe:

```
package io.github.bonigarcia;

import org.junit.jupiter.api.Test;

@Functional
class FunctionalTest {

    @Test
    void testOne() {
        System.out.println("Test 1");
    }

    @Test
    void testTwo() {
        System.out.println("Test 2");
```

```
    }

}
```

Fonte: Garcia, 2017.

Também podemos fazer anotações no nível do método. No teste a seguir, anotamos os diferentes testes (métodos) com diferentes anotações (@Load, @Stress, @Security e @Accessibility):

```
package io.github.bonigarcia;

import org.junit.jupiter.api.Test;

class NonFunctionalTest {

  @Test
  @Load
  void testOne() {
    System.out.println("Test 1");
  }

  @Test
  @Stress
  void testTwo() {
    System.out.println("Test 2");
  }

  @Test
  @Security
  void testThree() {
    System.out.println("Test 3");
  }

  @Test
  @Usability
  void testFour() {
    System.out.println("Test 4");
  }
```

}

Fonte: Garcia, 2017.

Saiba mais

Todos esses exemplos podem ser encontrados no repositório GitHub, acessando o *link* a seguir:

GITHUB. Bonigarcia. Disponível em: <https://github.com/bonigarcia/mastering-junit5>. Acesso em: 10 mar. 2023.

Encerramos esse assunto focado na ferramenta JUnit como forma de demonstrar o potencial da implementação de testes unitários para a linguagem de programação Java.

TEMA 4 – BOAS PRÁTICAS EM TESTES

Quando falamos em boas práticas, inevitavelmente somos remetidos às boas práticas em desenvolvimento de *software*. Fazendo uma transposição para a área de testes, também prezamos pelo mesmo conceito. Em testes, podemos classificar as boas práticas em três distintas categorias [Desikan,2007]:

1. Relacionado ao processo;
2. Pessoas relacionadas; e
3. Relacionado à tecnologia.

Vamos discutir cada uma das categorias nos subitens que seguem.

4.1 Boas práticas relacionando testes ao processo

Uma forte infraestrutura de processo e uma cultura de processo são necessárias para alcançar melhor previsibilidade e consistência. Modelos de processo como o CMMI podem fornecer uma estrutura para construir essa infraestrutura. A implementação de um modelo de processo formal que faça sentido para os negócios pode fornecer treinamento consistente para todos os colaboradores e garantir consistência na forma como as tarefas são executadas. Garantir processos amigáveis às pessoas cria pessoas amigáveis aos processos.

Integrar processos com tecnologia de forma inteligente é a chave para o sucesso de uma organização. Um banco de dados de processos, uma federação de informações sobre a definição e execução de vários processos, pode ser uma

adição valiosa ao repertório de ferramentas de uma organização. Quando esse banco de dados de processo é integrado a outras ferramentas, como repositório de defeitos, ferramenta SCM (Supply Chain Managment), a organização pode maximizar os benefícios.

4.2 Boas práticas relacionando testes a pessoas

A chave para o gerenciamento bem-sucedido é garantir que as equipes de teste e desenvolvimento funcionem bem. Esse relacionamento pode ser aprimorado ao criar-se um senso de propriedade nas metas abrangentes do produto. Embora metas individuais sejam necessárias para as equipes de desenvolvimento e teste, é muito importante chegar a um entendimento comum das metas gerais que definem o sucesso do produto como um todo. A participação das equipes de teste nesse processo de definição de metas e seu envolvimento inicial no processo geral de planejamento do produto podem ajudar a melhorar a gelificação necessária, a qual pode ser fortalecida mantendo as equipes de teste informadas sobre a tomada de decisões sobre o lançamento do produto e os critérios usados para o lançamento.

A rotação de tarefas entre suporte, desenvolvimento e teste também pode aumentar a integração entre as equipes. Essa rotação de tarefas pode ajudar as diferentes equipes a desenvolverem melhor empatia e apreciação dos desafios enfrentados nas funções de cada um e, assim, resultar em um melhor trabalho em equipe.

4.3 Boas práticas relacionando testes à tecnologia

Um repositório de defeitos pode ajudar na melhor automação das atividades de teste, auxiliando inclusive na escolha dos testes que provavelmente descobrirão defeitos. Mesmo que uma ferramenta de automação de teste em grande escala não esteja disponível, uma forte integração entre essas três ferramentas pode aumentar muito a eficácia do teste. Métricas e medições, densidade de defeitos, taxa de remoção de defeitos, bem como o cálculo dessas métricas são simplificados diante da integração entre essas ferramentas.

A automação de teste e ferramentas de automação eliminam o tédio e a rotina das funções de teste, além de aumentarem o desafio e o interesse nas

funções de teste. Apesar dos altos custos iniciais que podem incorrer, a automação de teste tende a gerar economias significativas de custos diretos de longo prazo, reduzindo o trabalho manual necessário para executar os testes. Há também benefícios indiretos em termos de menor desgaste dos engenheiros de teste, já que a automação de teste não apenas reduz a rotina, mas também traz um pouco de programação para o trabalho, que os engenheiros geralmente gostam.

Quando ferramentas de automação de teste são usadas, é útil integrar a ferramenta com repositório de defeitos e uma ferramenta SCM. Na verdade, a maioria das ferramentas de automação de teste fornece esses recursos, como o repositório de defeitos, enquanto se integra com ferramentas SCM comerciais.

Uma observação final sobre as melhores práticas. As três dimensões das melhores práticas não podem ser realizadas isoladamente. Uma boa infraestrutura de tecnologia deve ser adequadamente apoiada por uma infraestrutura de processo eficaz e ser executada por pessoas competentes. Essas práticas recomendadas são interdependentes, autossuficientes e se aprimoram mutuamente. Assim, a organização precisa ter uma visão holística dessas práticas e manter um bom equilíbrio entre as três dimensões.

TEMA 5 – *OVER THE SHOULDER*, TESTE COLABORATIVO

Quando um desenvolvedor termina a escrita de uma funcionalidade, esta deve ser mesclada com o projeto principal por meio de uma ferramenta de versionamento e colaboração de desenvolvimento. Antes de enviar para que a funcionalidade faça parte do projeto, deve-se separar uns 10 minutos, sozinho ou em pares com outro profissional (*designer*, analista de negócios etc.) para verificação.

Durante a verificação de ombro, numa tradução para *Over the shoulder*, o desenvolvedor mostrará cada critério de aceitação, o QA (analista de qualidade) pode pedir ao desenvolvedor para verificar outros casos que conhece e o *designer* pode validar se os projetos correspondem às especificações.

As verificações de ombro normalmente seriam para recursos grandes – os desenvolvedores poderiam registrar a tela ou pular a verificação de ombro para recursos menores. Se algo precisar ser corrigido como resultado, o desenvolvedor poderá fazê-lo antes da fusão. Se a verificação do ombro for aprovada, o desenvolvedor poderá mesclar o recurso.

Depois de mesclado, o QA ou testador testará esse recurso e poderá se concentrar em testes exploratórios e casos extremos, pois a maior parte da funcionalidade crítica já foi cotestada durante a verificação de ombro.

Dentre os benefícios na adoção de Testes Colaborativos, encontramos o compartilhamento de conhecimento e melhoria da compreensão. As verificações de ombro envolvem mais membros da equipe em qualidade e garantem que toda a equipe se concentre na produção de qualidade. Esse tipo de *coteste* leva a mais compartilhamento de conhecimento organicamente. Os desenvolvedores aprendem mais sobre testes, e os testadores aprendem mais sobre desenvolvimento e implementação.

Por vezes, nosso amedrontador *bug* não é exatamente um *bug*, mas um comportamento inesperado. Por exemplo, uma mensagem de um desenvolvedor sobre um *bug* específico ou até mesmo escrever um tíquete para ele pode ser detectado e resolvido rapidamente durante a *verificação do ombro*, mesmo antes de o código ser mesclado.

Esse tipo de verificação gera conhecimento entre os times e sobre o funcionamento do produto. A falha de comunicação pode gerar desacordos entre *designs* e desenvolvedores, mas ao utilizarmos verificações em equipe para averiguação de como o produto deve ser e funcionar, evitamos a criação de alguns chamados por conta de um *bug*.

Critérios funcionais são testados antes do lançamento, caracterizando um excelente benefício também. Com verificações de ombro, a funcionalidade crítica foi cotestada antes do final do ciclo de desenvolvimento de *software*, o que dá aos clientes alguma certeza sobre a estabilidade dos recursos na próxima compilação. Em algumas equipes, o Product Owner, BA ou QA do cliente também podem participar das verificações de ombro. Isso ajuda a alinhar/confirmar os requisitos do cliente no recurso criado. Incluir os clientes nas verificações de ombro permite que eles se envolvam mais no processo de desenvolvimento, em vez de apenas planejar o *sprint* no início e testar a aceitação do usuário no final.

No geral, os clientes devem ter um produto melhor no final, pois os recursos passaram por mais portões de qualidade. Além disso, os clientes estão mais cientes do progresso da entrega em avanços, como atrasos em um recurso devido a *bugs* críticos ou critérios de aceitação perdidos durante verificações de ombro.

Finalmente, as taxas de entrega são aprimoradas, gerando outro considerável benefício. Quando a funcionalidade crítica é testada no início da *sprint*, ela ajuda a expor quaisquer riscos à entrega no início. Uma das maiores perdas de tempo é quando o controle de qualidade precisa reabrir um histórico de usuário devido a um critério de aceitação com falha ou ausente. Isso é facilmente detectado nas verificações do ombro e reduz as idas e vindas mais tarde à *sprint*. Posteriormente, há mais alinhamento nos requisitos e nos recursos que são construídos como QA, *designers*, BA e desenvolvedores estão se comunicando com frequência.

Ter *designers* nessas verificações de ombro é ainda mais benéfico e importante se o aplicativo for pensado em termos de *design*. Em um projeto, os desenvolvedores tendem a enviar capturas de tela de novas telas que estão implementando atualmente como um rascunho para obter *feedback* de *design* antecipadamente.

O processo de testes deve ser implementado para a busca constante de melhores resultados, e os testes colaborativos/verificações de ombro são um importante caminho. Conversar com todo o time e compreender as preocupações quanto ao uso de testes colaborativos são benefícios após algumas semanas de teste. Documentar o processo de verificação por meio de pequenos resumos auxilia no destaque de aspectos importantes. Reuniões curtas (5 a 10 minutos) entre desenvolvedores, analistas de negócios, *designers* e analistas de qualidade podem fortalecer o processo de testes colaborativos e o projeto de forma geral. Finalmente, problemas críticos detectados desde o início do processo reduzem os ciclos de *feedback*. Qualidade é responsabilidade de todos os times, e o teste colaborativo traz muitas vantagens com suas dinâmicas de trabalho.

FINALIZANDO

Não importa quantos e quais técnicas ou ferramentas utilizemos num processo de qualidade de *software*, mais especificamente no momento de testes, validações e verificações. O que importa é o quanto otimizamos nossos processos de testes e de desenvolvimento de *software*. O que importa é a qualidade que pretendemos alcançar tanto a nível de processos quanto a nível de produto.

O que vai guiar nosso sucesso na adoção de processos de testes é o

engajamento real que teremos de todos os participantes dentro dos times que atuam em parceria.

O TDD aliado do XP tem trazido grande contribuição para a disseminação da necessidade de testes desde a concepção da menor funcionalidade, classe ou outros componentes de *software*. O ideal é que se inicie aos poucos com a cultura de qualidade e testes e ao longo do tempo veremos tanto profissionais da área de testes e qualidade quanto do desenvolvimento de *software* empenhados na busca na excelência em qualidade de *software*.

Ninguém fala que é fácil a implementação de processos de testes; justamente o contrário: temos um esforço inicial bastante grande, no qual muitos precisam aprender e aplicar os conhecimentos ainda novos. Mas quando conseguimos amadurecer em relação a esses processos, certamente problemas de falhas de *software* já não absorvem e desperdiçam orçamentos, tempo e recursos humanos.

REFERÊNCIAS

BECK, K. **Test-driven development**. Boston, MA: Addison-Wesley, 2003.

BROWN, E. **Web development with node and express**. Sebastopol, CA: O'Reilly Media, Inc., 2014.

DESIKAN, S. **Software testing**: principles and practices. Sebastopol, California: O'Reilly Media, Inc., 2007.

GARCIA, B. **Mastering software testing with JUnit 5**. Birmingham: Packt, 2017.

HAMBLING, B. et al. **Software testing**: an ISTQB-BCS certified tester foundation guide. 3. ed. Swindon, UK: BCS Learning & Development Ltd. 2015.

JUNIT. JUnit – Teste de Unidade. Disponível em: <http://junit.wikidot.com/>. Acesso em: 10 mar. 2023.

LAPORTE, C.; APRIL, A. **Software quality assurance**. New York: IEEE Press Wiley, 2018.

LATINO, R. **Root cause analysis**: improving performance for bottom-line results. CRC Press, 2011.

LEWIS, W. E. **Software testing and continuous quality improvement**. 3. ed. Boca Raton, FL: Taylor & Francis Group, LLC, 2009.

MOHRAN, G. **Full stack testing**. Sebastopol, California: O'Reilly Media, Inc., 2022.

PRESSMAN, R. S. **Engenharia de *software***: uma abordagem profissional. 7. ed. Porto Alegre: AMGH, 2011.

SOMMERVILLE, I. **Engenharia de *software***. São Paulo. Pearson Education do Brasil, 2018.